The `goplumb` package for manipulating `plumb` messages

(version 0.4.0)

Alexander Sychev (santucco@gmail.com)

1. Introduction. In a great operating system `Plan 9` there is a `plumber -` a filesystem for interprocess messaging. The `goplumb` package is implemented to manipulate such messages. The main target of the package is support of `plumber` from `Plan 9 from User Space` project http:// swtch.com/plan9port/.

2. Implementation.

```
    // Package goplumb provides interface to plumber - interprocess messaging from Plan 9.
package goplumb
import(
  ⟨ Imports 6 ⟩
)
type(
  ⟨ Types 4 ⟩
)
var(
  ⟨ Variables 9 ⟩
)
```

3. Let's describe a begin of a test for the package. The `plumber` will be be started for the test.

```
⟨goplumb_test.go 3⟩ ≡
  package goplumb
  import(
    "os/exec"
    "testing"
    "bytes"
    "time"
    "9fans.net/go/plan9"
    "9fans.net/go/plan9/client"
    ⟨Test specific imports 22⟩
  )
  const rule = `type␣is␣text\nsrc␣is␣Test\nplumb␣to␣goplumb\n`
  var fs *client.Fsys
  func prepare(t *testing.T){
        // checking for a running plumber instance
    var err error
    fs, err = client.MountService("plumb")
    if err ≡ nil {
      t.Log("plumber␣started␣already")
    } else {
          // start plumber
      cmd := exec.Command("plumber")
      err = cmd.Run()
      if err ≠ nil {
        t.Fatal(err)
      }
      t.Log("plumber␣is␣starting,␣wait␣a␣second")
      time.Sleep(time.Second)
    }
    fs, err = client.MountService("plumb")
    if err ≠ nil {
      t.Fatal(err)
    }
          // setting a rule for the test
    f, err := fs.Open("rules", plan9.OWRITE)
    if err ≠ nil {
      t.Fatal(err)
    }
    defer f.Close()
    _, err = f.Write([]byte(rule))
    if err ≠ nil {
      t.Fatal(err)
    }
  }
  func compare(m1 *Message, m2 *Message) bool{
    if m1.Src ≠ m2.Src ∨
      m1.Dst ≠ m2.Dst ∨
      m1.Wdir ≠ m2.Wdir ∨
      m1.Type ≠ m2.Type ∨
      len(m1.Attr) ≠ len(m2.Attr) {
```

```
    return false
  }
  for n, v := range m1.Attr {
    if m2.Attr[n] ≠ v {
      return false
    }
  }
  return bytes.Compare(m1.Data, m2.Data) ≡ 0
}
⟨ Test routines 13 ⟩
```

4. At first let's describe *Message* structure. Actually it is almost the same like the original `Plan 9` C-struct. *Src* is a source of a message; *Dst* is a destination; *Wdir* is a working directory; *Type* is a type of the message, usually *text*; *Attr* is a slice of attributes of the message where an attribute is a pair of *name* = *value*; *Data* is a binary data of the message.

```
⟨ Types 4 ⟩ ≡
      // Message desribes a plumber message.
  Message struct{
    Src string
    Dst string
    Wdir string
    Type string
    Attr Attrs
    Data []byte
  }
```

See also sections 5 and 7.

This code is used in section 2.

5.

```
⟨ Types 4 ⟩ +≡
      // Attrs is a map of an attribute of a plumber message.
  Attrs map[string]string
```

6. A *Plumb* is a top-level structure. It contains a pointer to *os.File*, which is a port in `plumber`'s file system. All fields of the *Plumb* are unexported.

```
⟨ Imports 6 ⟩ ≡
  "9fans.net/go/plan9"
  "9fans.net/go/plan9/client"
  "os"
```

See also sections 10, 15, 17, 20, and 25.

This code is used in section 2.

7. ⟨ Types 4 ⟩ +≡

```
  Plumb struct{
    f ∗client.Fid
    ⟨ Other members of Plumb 30 ⟩
  }
```

8. Open. At first if *port* is not an absolute filename, a slash is added if neccessary at the end of *port*. Then a file is opened with specified *omode*.

9. At first we have to mount `plumber` namespace

```
⟨ Variables 9 ⟩ ≡
  fsys *client.Fsys
  sp *Plumb
  rp *Plumb
```

This code is used in section 2.

10.

```
⟨ Imports 6 ⟩ +≡
  "sync"
```

11.

```
⟨ Mount plumber namespace 11 ⟩ ≡
  {
    var err error
    new(sync.Once).Do(func(){
      fsys, err = client.MountService("plumb")
    })
    if err ≠ nil {
      return nil, err
    }
  }
```

This code is used in section 12.

12.

```
      // Open opens a specified port with a specified omode and returns *Plumb or error
  func Open(port string, omode uint8) (*Plumb, error){
    ⟨ Mount plumber namespace 11 ⟩
    var p Plumb
    var err error
    if p.f, err = fsys.Open(port, omode); err ≠ nil {
      return nil, err
    }
    return &p, nil
  }
```

13. Let's test *Open* function.

⟨ Test routines 13 ⟩ ≡

```
func TestOpen(t *testing.T){
  prepare(t)
  var err error
  if sp, err = Open("send", plan9.OWRITE);  err ≠ nil {
    t.Fatal(err)
  }
  if rp, err = Open("goplumb", plan9.OREAD);  err ≠ nil {
    t.Fatal(err)
  }
}
```

See also sections 23, 27, 32, and 33.

This code is used in section 3.

14. Send. A *message* is packed and is written to the file.

```
    // Send sends a message and returns error if any.
func (this * Plumb) Send(message * Message) error{
  if this ≡ nil ∨ this.f ≡ nil ∨ message ≡ nil {
    return os.ErrInvalid
  }
  b := Pack(message)
      // a workaround: plumber can't receive a message with length more that 8192 − plan9.IOHDRSIZE
  for len(b)⟩0 {
    c := 8192 − plan9.IOHDRSIZE
    if len(b)⟨c {
      c = len(b)
    }
    c, err := this.f.Write(b[:c])
    if err ≠ nil {
      return err
    }
    b = b[c:]
  }
  return nil
}
```

15. Pack. All the fields of a *message* are packed like a strings delimeted by newlines.

⟨ Imports 6 ⟩ +≡

```
  "fmt"
```

16.

```
      // Pack packs a message to []byte.
  func Pack(message *Message) []byte{
    s := fmt.Sprintf("%s\n%s\n%s\n%s\n%s\n%d\n",
      message.Src, message.Dst,
      message.Wdir, message.Type,
      PackAttr(message.Attr),
      len(message.Data))
    b := append([]byte{}, []byte(s)...)
    return append(b, message.Data...)
  }
```

17. PackAttr. Attributes *attr* are packed like pairs *Name* = *Value* delimeted by spaces. *Value* can be quoted if it is neccessary.

```
⟨ Imports 6 ⟩ +≡
  "strings"
```

18.

```
      // PackAttr packs attr to string. If an attribute value contains a white space,
      // a quote or an equal sign the value will be quoted.
  func PackAttr(attr Attrs) string{
    var s string
    first := true
    for n, v := range attr {
      if ¬first {
        s += "␣"
      }
      first = false
      if strings.ContainsAny(v, "␣'=\t") {
        s += fmt.Sprintf("%s='%s'", n, strings.Replace(v, "'", "''", −1))
      } else {
        s += fmt.Sprintf("%s=%s", n, v)
      }
    }
    return s
  }
```

19. SendText. A message is composed from *Src* = *src*, *Dst* = *dst*, *Wdir* = *wdir* and *Type* = *text*

```
    // SendText sends a text-only message; it assumes Type is text and Attr is empty.
    // SendText returns error if any.
func (this *Plumb) SendText(src string, dst string, wdir string, data string) error{
  m := &Message{
    Src: src,
    Dst: dst,
    Wdir: wdir,
    Type: "text",
    Data: []byte(data)}
  return this.Send(m)
}
```

20. Recv. At most 8192 bytes are read for the first time. Then *UnpackPartial* is used to unpack a message. If the message too big *b* is reallocated for needed size, last part of the message is read and the message is unpacked.

```
⟨ Imports 6 ⟩ +≡
  "errors"
  "io"
```

21.

```
      // Recv returns a pointer to a received message *Message or error.
  func (this * Plumb) Recv() (*Message, error){
    if this ≡ nil ∨ this.f ≡ nil {
      return nil, os.ErrInvalid
    }
    b := make([]byte, 8192)
    n, err := this.f.Read(b)
    if err ≠ nil {
      return nil, err
    }
    m, r := UnpackPartial(b[:n])
    if m ≠ nil {
      return m, nil
    }
    if r ≡ 0 {
      return nil, errors.New("buffer␣too␣small")
    }
    if r⟩len(b) − n {
      b1 := make([]byte, r + n)
      copy(b1, b)
      b = b1
    } else {
      b = b[:n + r]
    }
    _, err = io.ReadFull(this.f, b[n:])
    if err ≠ nil {
      return nil, err
    }
    m, r = UnpackPartial(b)
    if m ≠ nil {
      return m, nil
    }
    return nil, errors.New("buffer␣too␣small")
  }
```

22. Let's test *Send* and *Recv* functions.

```
⟨ Test specific imports 22 ⟩ ≡
  "errors"
```

This code is used in section 3.

23. ⟨ Test routines 13 ⟩ +≡

```
func TestSendRecv(t *testing.T){
  var m Message
  m.Src = "Test"
  m.Dst = "goplumb"
  m.Wdir = "."
  m.Type = "text"
  m.Attr = make(Attrs)
  m.Attr["attr1"] = "value1"
  m.Attr["attr2"] = "value2"
  m.Attr["attr3"] = "value␣=␣'3\t"
  m.Data = []byte("1234567890")
  if err := sp.Send(&m); err ≠ nil {
    t.Fatal(err)
  }
  t.Logf("message␣%#v␣has␣been␣sent\n", m)
  r, err := rp.Recv()
  if err ≠ nil {
    t.Fatal(err)
  }
  t.Logf("message␣%#v␣has␣been␣received\n", *r)
  if ¬compare(r, &m) {
    t.Fatal(errors.New("messages␣is␣not␣matched"))
  }
}
```

24. Unpack. *Unpack* just recalls *UnpackPartial* and ignores a rest of a message if the message is too big.

```
        // Unpack return a pointer to an unpacked message *Message.
  func Unpack(b []byte) *Message{
    m, _ := UnpackPartial(b)
    return m
  }
```

25. UnpackPartial.

⟨ Imports 6 ⟩ +≡
```
  "bytes"
  "strconv"
```

26.

```
      // UnpackPartial helps to unpack messages splited in peaces.
      // The first call to UnpackPartial for a given message must be sufficient to unpack
      // the header; subsequent calls permit unpacking messages with long data sections.
      // For each call, b contans the complete message received so far.
      // If the message is complete, a pointer to the resulting message m will be returned,
      // and a number of remainings bytes r will be set to 0.
      // Otherwise m will be nil and r will be set to the number of bytes
      // remaining to be read for this message
      // to be complete (recall that the byte count is in the header).
      // Those bytes should be read by the caller, placed at location b[r:],
      // and the message unpacked again.
      // If an error is encountered, m will be nil and r will be zero.
  func UnpackPartial(b []byte) (m *Message, r int){
    bb := bytes.Split(b, []byte{'\n'})
    if len(bb)<6 {
      return nil, 0
    }
    m = &Message{Src: string(bb[0]), Dst: string(bb[1]), Wdir: string(bb[2]), Type:
    string(bb[3]), Attr: UnpackAttr(string(bb[4]))}
    n, err := strconv.Atoi(string(bb[5]))
    if err ≠ nil {
      return nil, 0
    }
    i := 0
    for j := 0; j<6; j++ {
      i += len(bb[j]) + 1
    }
    if r = n − (len(b) − i); r ≠ 0 {
      return nil, r
    }
    m.Data = make([]byte, n)
    copy(m.Data, b[i:i + n])
    return m, 0
  }
```

27. Let's test *Send* and *Recv* functions with a big message.

```
⟨ Test routines 13 ⟩ +≡
  func TestSendRecvBigMessage(t ∗ testing.T){
    var m Message
    m.Src = "Test"
    m.Dst = "goplumb"
    m.Wdir = "."
    m.Type = "text"
    m.Attr = make(Attrs)
    m.Attr["attr1"] = "value1"
    m.Attr["attr2"] = "value2"
    m.Attr["attr3"] = "value␣=␣'3\t"
    m.Data = make([]byte, 0, 9000)
    for i := 0; i⟨900; i++ {
      m.Data = append(m.Data, []byte("1234567890")...)
    }
    if err := sp.Send(&m); err ≠ nil {
      t.Fatal(err)
    }
    t.Logf("message␣%#v␣has␣been␣sent\n", m)
    r, err := rp.Recv()
    if err ≠ nil {
      t.Fatal(err)
    }
    t.Logf("message␣%#v␣has␣been␣received\n", ∗r)
    if ¬compare(r, &m) {
      t.Fatal(errors.New("messages␣is␣not␣matched"))
    }
  }
```

28. UnpackAttr. *UnpackAttr* unpacks attributes from s, unquotes values if it is neccessary.

```
    // UnpackAttr unpack the attributes from s to Attrs
func UnpackAttr(s string) Attrs{
  attrs := make(Attrs)
  for i := 0; i⟨len(s); {
    var n, v string
    for ; i⟨len(s) ∧ s[i] ≠ '='; i++ {
      n += s[i:i + 1]
    }
    i++
    if i ≡ len(s) {
      break
    }
    if s[i] ≡ '\'' {
      i++
      for ; i⟨len(s); i++ {
        if s[i] ≡ '\'' {
          if i + 1 ≡ len(s) {
            break
          }
          if s[i + 1] ≠ '\'' {
            break
          }
          i++
        }
        v += s[i:i + 1]
      }
      i++
    } else {
      for ; i⟨len(s) ∧ s[i] ≠ '␣'; i++ {
        v += s[i:i + 1]
      }
    }
    i++
    attrs[n] = v
  }
  return attrs
}
```

29. Close. *Close* just closes an underlying f.

```
    // Close closes the plumbing connection.
func (this * Plumb) Close(){
  if this ≠ nil ∧ this.f ≠ nil {
    this.f.Close()
    this.f = nil
  }
}
```

30. MessageChannel.

```
⟨ Other members of Plumb 30 ⟩ ≡
  ch chan *Message
```

This code is used in section 7.

31.

```
      // MessageChannel returns a channel of *Message with a buffer size
      // from which messages can be read or error.
      // First call of MessageChannel starts a goroutine to read messages put them to the channel.
      // Subsequent calls of EventChannel will return the same channel.
  func (this *Plumb) MessageChannel(size int) (← chan *Message, error){
    if this ≡ nil ∨ this.f ≡ nil {
      return nil, os.ErrInvalid
    }
    if this.ch ≠ nil {
      return this.ch, nil
    }
    this.ch = make(chan *Message, size)
    go func(ch chan ← *Message){
      for m, err := this.Recv(); err ≡ nil; m, err = this.Recv() {
        ch ← m
      }
      close(ch)
    }(this.ch)
    return this.ch, nil
  }
```

32. A test of *MessageChannel* function.

```
⟨ Test routines 13 ⟩ +≡
 func TestMessageChannel(t *testing.T){
   var m Message
   m.Src = "Test"
   m.Dst = "goplumb"
   m.Wdir = "."
   m.Type = "text"
   m.Attr = make(Attrs)
   m.Attr["attr1"] = "value1"
   m.Attr["attr2"] = "value2"
   m.Attr["attr3"] = "value␣=␣'3\t"
   m.Data = []byte("1234567890")
   ch, err := rp.MessageChannel(0)
   if err ≠ nil {
     t.Fatal(err)
   }
   if err := sp.Send(&m); err ≠ nil {
     t.Fatal(err)
   }
   t.Logf("message␣%#v␣has␣been␣sent\n", m)
   time.Sleep(time.Second)
   rm, ok :=← ch
   if ¬ok {
     t.Fatal(errors.New("messages␣channel␣is␣closed"))
   }
   t.Logf("message␣%#v␣has␣been␣received\n", *rm)
   if ¬compare(rm, &m) {
     t.Fatal(errors.New("messages␣is␣not␣matched"))
   }
 }
```

33. A test of *Close* function.

```
⟨ Test routines 13 ⟩ +≡
 func TestClose(t *testing.T){
   rp.Close()
   sp.Close()
 }
```

34. Index.

The `goplumb` package for manipulating `plumb` messages

(version 0.4.0)